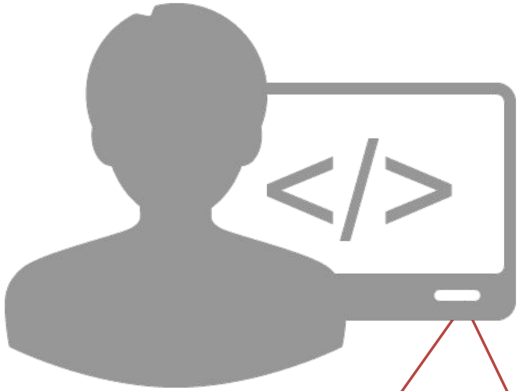
Unit-6

Software Coding & Testing

✓ Outline

- Code Review
- Software Documentation
- Testing Strategies
- Testing Techniques and Test Case
- Test Suites Design
- Testing Conventional Applications
- Testing Object Oriented Applications
- Testing Web and Mobile Applications

Coding Standards



Good software development organizations normally require their **programmers** to **adhere** to some **well-defined** and standard style of **coding** called coding standards.

- Most software development organizations formulate their **own coding standards** that suit them most, and **require** their **engineers to follow** these standards strictly
- The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

A coding standard gives a uniform appearance to the codes written by different engineers.

It enhances code understanding.

It encourages good programming practices.

Coding Standards Cont.

A coding standard lists **several rules** to be **followed** such as, the **way variables** are to be **named**, the **way** the **code** is to **be laid out**, **error** return **conventions**, etc.

The following are some representative coding standards

1 Rules for limiting the use of global

These rules list what types of data can be declared global and what cannot.

2 Naming conventions for global & local variables & constant identifiers

A possible naming convention can be that **global variable** names always **start with a capital letter**, **local variable** names are made of **small letters**, and **constant names** are **always capital** letters.

3 Contents of the headers preceding codes for different modules

- The **information contained in the headers** of different modules **should be standard** for an organization.
- The **exact format** in which the header information is organized in the header can **also be specified**

The following are some standard header data

Module Name | **Creation Date** | **Author's Name**

Modification history | **Synopsis of the module**

Global variables accessed/modified by the module

Different functions supported, along with their input/output parameters

Coding Standards Cont.

Sample Header

```
/*
 * MyClass <br>
 * This class is merely for illustrative purposes. <br>
 *Revision History:<br>
 * 1.1 – Added javadoc headers <br>
 * 1.0 - Original release<br>
 * @author P.U. Jadeja
 * @version 1.1, 12/02/2018
 */
public class MyClass  {
    . . .
}
```

❹ Error return conventions and exception handling mechanisms

- The **way error** conditions are **reported** by different functions in a program are handled should be standard within an organization.
- For **example**, different functions **while encountering an error** condition should **either return a 0 or 1** consistently.

Coding guidelines

The following are some representative coding guidelines

Do not use a coding style that is **too clever** or **too difficult** to **understand**

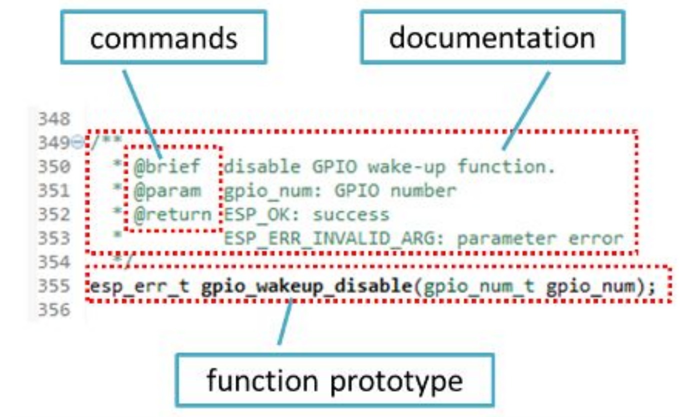
Do **not use** an **identifier** for **multiple purposes**

The **code** should be **well-documented**

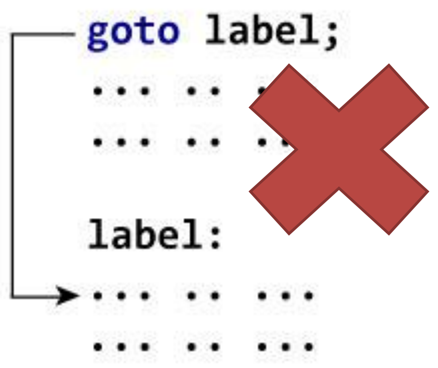
The **length of** any **function** should **not exceed 10** source **lines**

Do **not use goto** statements

Well Documented



```
/**
 * @brief  disable GPIO wake-up function.
 * @param  gpio_num: GPIO number
 * @return ESP_OK: success
 *         ESP_ERR_INVALID_ARG: parameter error
 */
esp_err_t gpio_wakeup_disable(gpio_num_t gpio_num);
```



```
goto label;
... .. ...
... .. ...

label:
... .. ...
... .. ...
```

Do not use goto

Avoid obscure side effects

- The side **effects of a function call** include ***modification of parameters passed by reference***, ***modification of global variables***, and I/O operations.
- An **obscure side effect** is one that **is not obvious from a casual examination** of the code.
- Obscure side effects **make it difficult to understand** a piece of code.
- For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

Software Faults

- Quite inevitable (unavoidable)
- Many reasons

Software systems with large number of states

Complex formulas, activities, algorithms

Customer is often unclear of needs

Size of software

Number of people involved

Types of Faults

Algorithmic	Logic is wrong Code reviews
Syntax	Wrong syntax; typos Compiler
Computation/ Precision	Not enough accuracy
Documentation	Misleading documentation
Stress/Overload	Maximum load violated
Capacity/Boundary	Boundary cases are usually special cases
Timing/Coordination	Synchronization issues Very hard to replicate
Throughput/Performance	System performs below expectations
Recovery	System restarted from abnormal state
Hardware & related software	Compatibility issues
Standards	Makes for difficult maintenance

Software Quality

Software Quality remains an issue

Who is to blame?

Customers blame developers

Arguing that careless practices lead to low-quality software

Developers blame Customers & other stakeholders

Arguing that irrational delivery dates and continuous stream of changes force the to deliver software before it has been fully validated

Who is Right? Both – and that's the problem

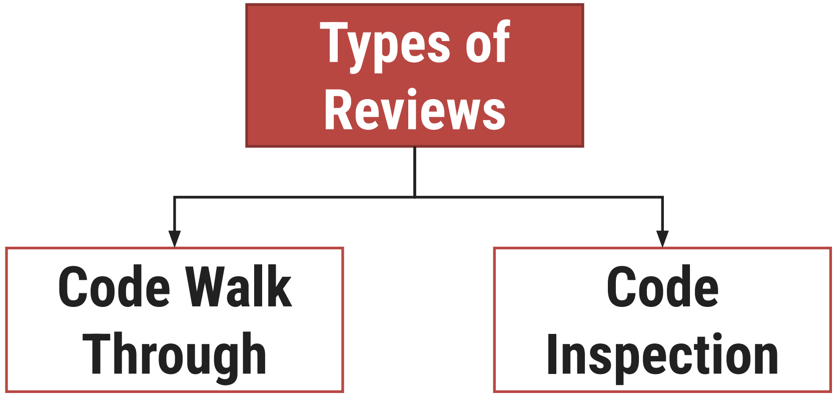
Code Review

Code Walk Through

Code Inspection

Code Review

- Code Review is carried out **after the module is successfully compiled** and all the syntax errors have been eliminated.
- Code Reviews are extremely **cost-effective strategies** for **reduction in coding errors** and to produce high quality code.



Few classical programming errors

- Use of uninitialized variables
- Jumps into loops
- Nonterminating loops
- Incompatible assignments
- Array indices out of bounds
- Improper storage allocation and deallocation
- Mismatches between actual and formal parameter in procedure calls
- Use of incorrect logical operators or incorrect precedence among operators
- Improper modification of loop variables

Code Review

Code Walk Through & Code Inspection

Code Walk Through

- Code walk through is an **informal code analysis** technique.
- The main **objectives** of the walk through are **to discover** the **algorithmic** and **logical errors** in the **code**.
- A **few members** of the development **team** are given the **code** few days before the walk through meeting to read and understand code.
- Each **member selects some test cases** and **simulates execution** of the code **by hand**
- The members **note down their findings** to **discuss** these **in a walk through meeting** where the coder of the module is present.

Code Inspection

- The **aim** of **Code Inspection** is to **discover** some **common types** of **errors** caused **due to improper programming**.
- In other words, during Code Inspection **the code is examined** for the **presence** of certain **kinds of errors**.
  - For instance, consider the classical error of writing a procedure that modifies a parameter while the calling routine calls that procedure with a constant actual parameter.
  - It is more likely that **such an error will be discovered by looking for these kinds of mistakes in the code**.
- In addition, **commitment to coding standards** is also **checked**.

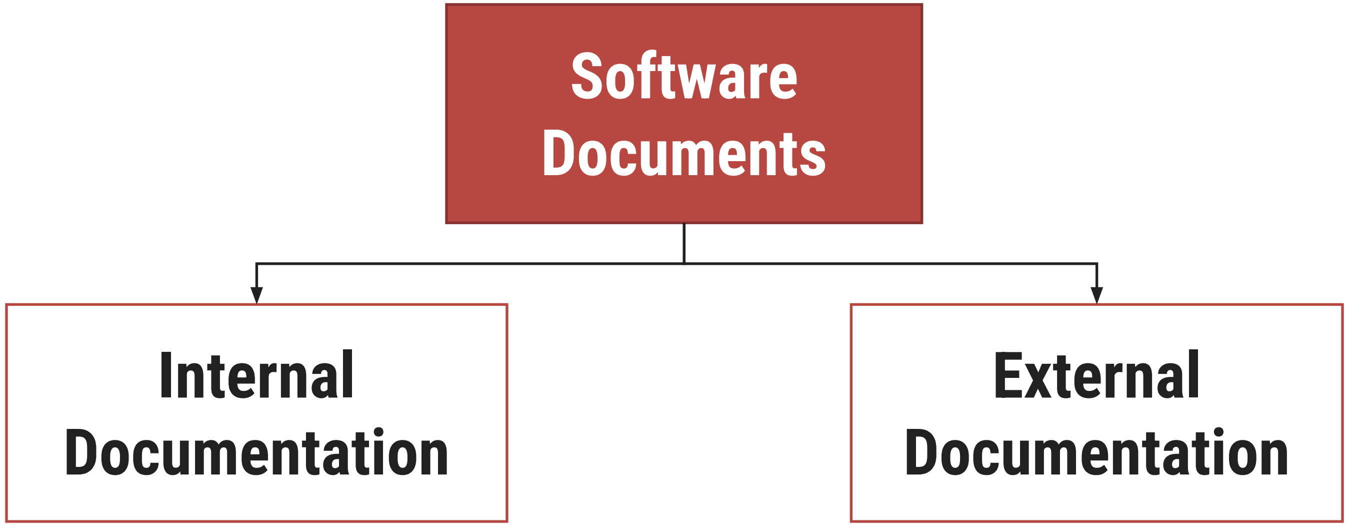
Software Documentation

- When various kinds of software products are developed, various kinds of **documents** are also developed as part of any software engineering process e.g..

Users’ manual | Design documents | Test documents | Installation manual

Software requirements specification (SRS) documents, etc

- Different **types of software documents** can broadly be classified into the following:



Software Documentation Cont.

Internal & External Documentation

Internal Documentation

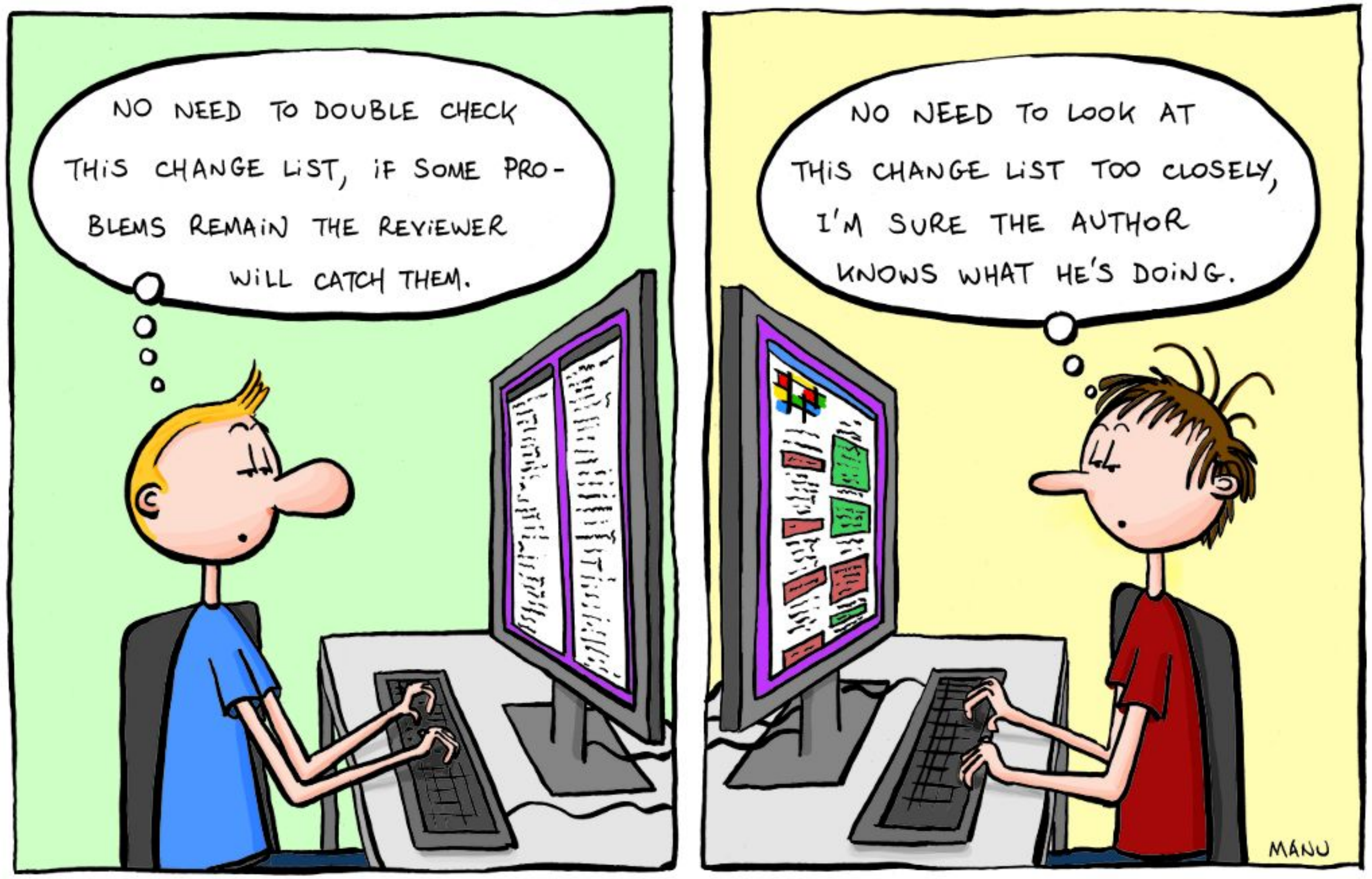
- It is the **code perception features** provided as part of the source code.
- It is provided through appropriate **module headers and comments** embedded in the source code.
- It is also provided through the useful **variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types**, etc.
- Even when code is carefully commented, **meaningful variable names** are still more helpful in understanding a piece of code.
- Good organizations ensure good internal documentation by appropriately formulating their coding standards and guidelines.

External Documentation

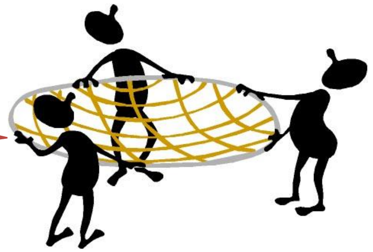
- It is provided through various types of **supporting documents**
  - such as users' manual
  - software requirements specification document
  - design document
  - test documents, etc.
- A systematic software development style ensures that all these documents are **produced in an orderly fashion**.

Software Testing

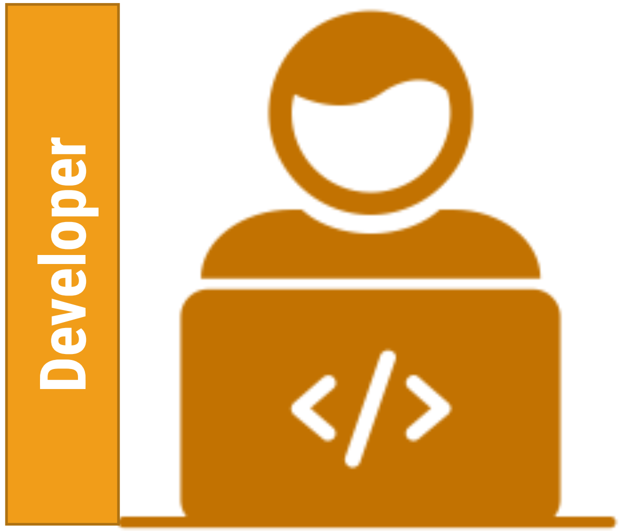
Testing is the **process** of exercising a program with the specific **intent of finding errors** prior to delivery to the end user.


NO NEED TO DOUBLE CHECK THIS CHANGE LIST, IF SOME PROBLEMS REMAIN THE REVIEWER WILL CATCH THEM.
NO NEED TO LOOK AT THIS CHANGE LIST TOO CLOSELY, I'M SURE THE AUTHOR KNOWS WHAT HE'S DOING.
MANU

Don't view testing as a **"safety net"** that will catch all errors that occurred because of weak software engineering practice.



Who Test the Software



Understands the system but, will test "gently"
and, is driven by "delivery"

Must learn about the system, but, will attempt to break it
and, is driven by quality

Testing without plan is of no point
It wastes time and effort

Testing need a strategy
Dev team needs to work with Test team, “Egoless Programming”

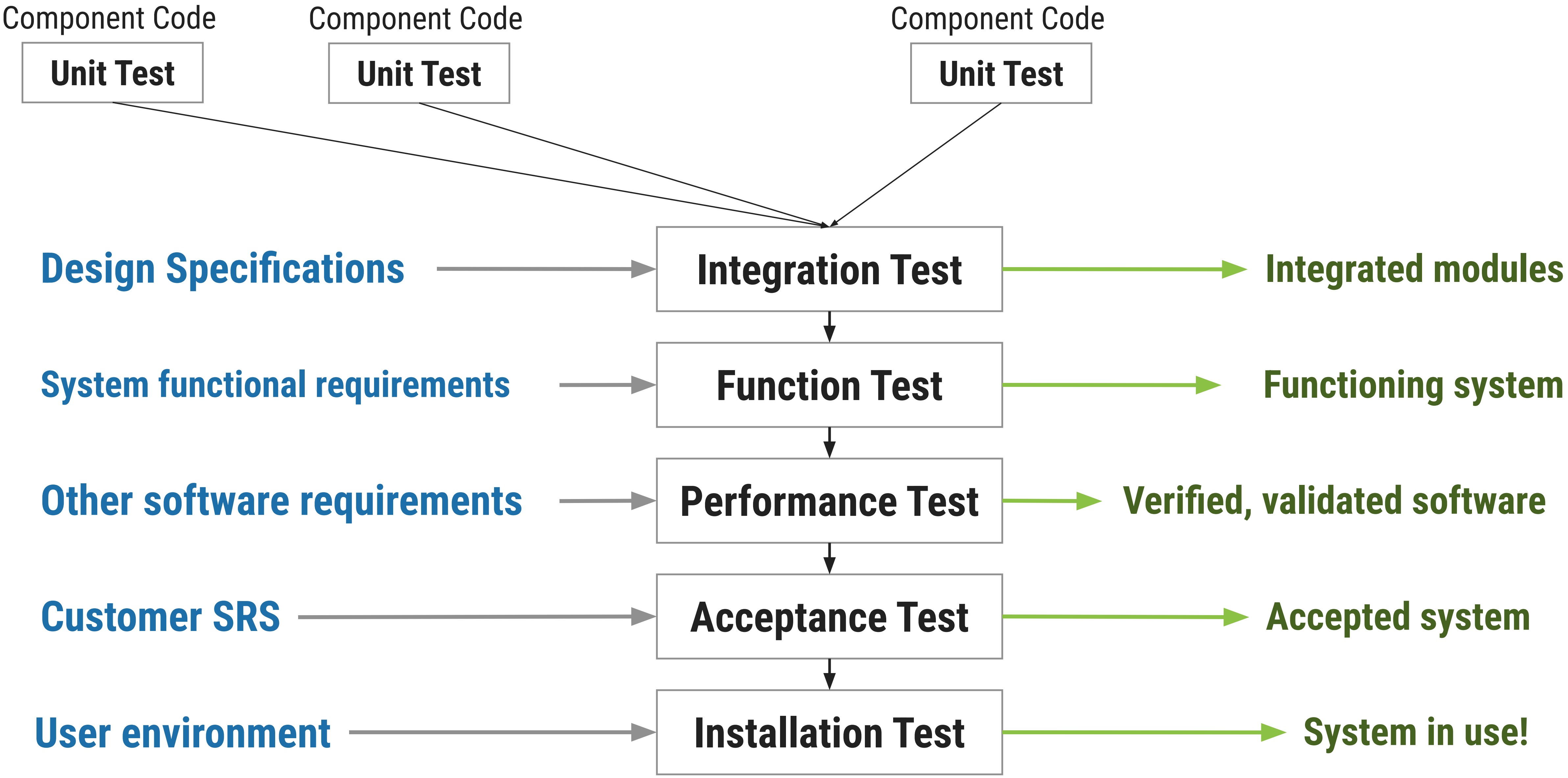
Who Test the Software?

<Developer>

OR

[Tester]

When to Test the Software?



Verification & Validation

Verification

Are we building the product right?

The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.

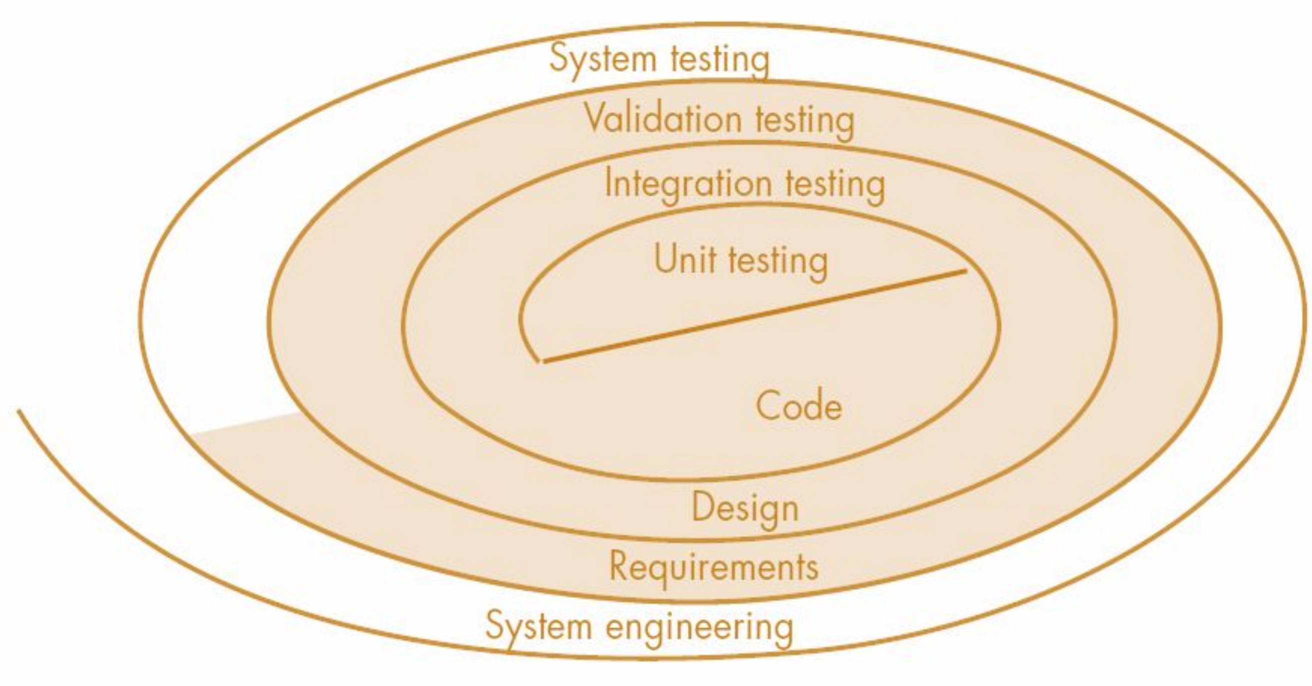
Validation

Are we building the right product?

The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.

Verification	Validation
Process of **evaluating** products of a **development phase** to find out whether they meet the specified requirements.	Process of evaluating software **at the end of the development** to determine whether software meets the customer expectations and requirements.
Activities involved: **Reviews**, **Meetings** and **Inspections**	Activities involved: **Testing** like black box testing, white box testing, gray box testing
Carried out by **QA team**	Carried out by **testing team**
Execution of code is **not comes** under Verification	**Execution** of code is **comes** under Validation
Explains whether the **outputs are according to inputs** or not	**Describes** whether the **software** is **accepted by** the **user** or not
Cost of **errors** caught is **less**	**Cost** of **errors** caught is **high**

Software Testing Strategy



- Unit Testing
- Integration Testing
- Validation Testing
- System Testing

Unit Testing

It **concentrate** on **each unit** of the software as **implemented in source code**

It **focuses** on each **component individual**, ensuring that it functions properly as a unit.

Software Testing Strategy Cont.

Integration Testing

It **focus** is on **design** and **construction** of **software architecture**

Integration testing is the **process** of **testing** the **interface between two software units** or modules

Validation Testing

Software is **validated against requirements** established as a part of requirement modeling

It give **assurance** that software meets all informational, functional, behavioral and performance requirements

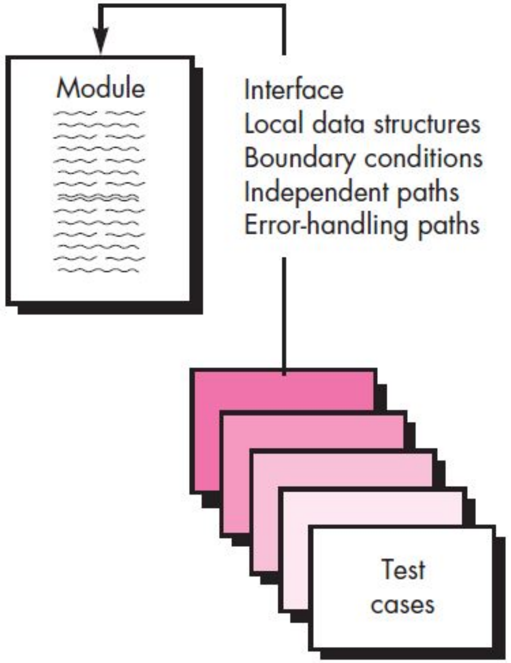
System Testing

The **software** and **other software elements** are **tested as a whole**

Software once validated, must be combined with other system elements e.g. hardware, people, database etc…

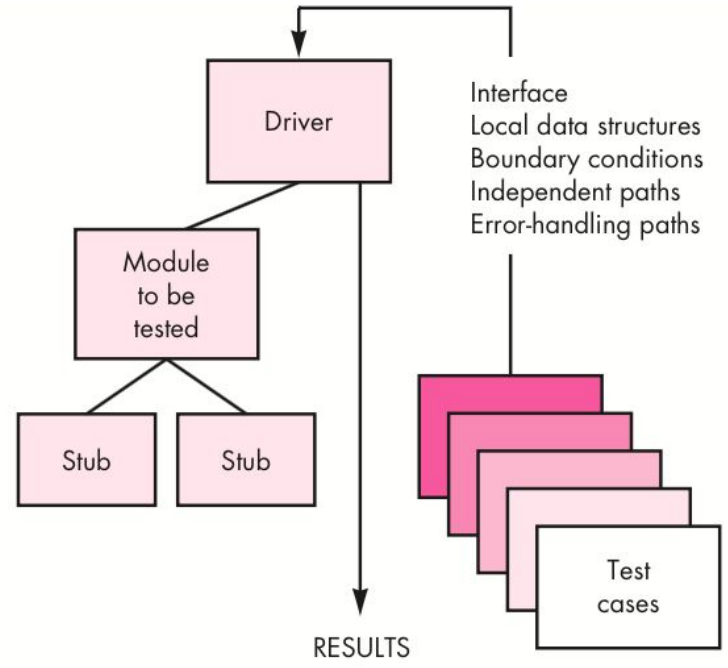
It verifies that all elements mesh properly and that overall system function / performance is achieved.

Unit Testing



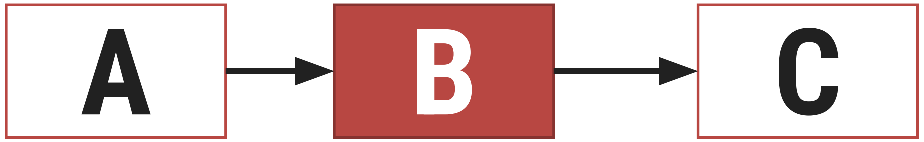
- Unit is the **smallest part of a software** system which is testable.
- It may include code files, classes and methods which can be tested individually for correctness.
- Unit Testing **validates small building block** of a complex system before testing an integrated large module or whole system
- The **unit test focuses** on the **internal processing logic** and **data structures** within the boundaries of a component.
- The module is tested to ensure that **information properly flows into** and **out** of the program unit
- **Local data structures** are examined to ensure that **data stored temporarily** maintains its **integrity** during execution
- All **independent paths** through the control structures are **exercised** to **ensure** that **all statements in module** have been **executed** at least **once**
- **Boundary conditions** are **tested** to **ensure** that the module **operates properly** at **boundaries** established to limit or restricted processing
- All **error handling** paths are **tested**

Diver & Stub (Unit Testing)



Component-testing (**Unit Testing**) may be **done** in **isolation** from rest of the system

In such case the **missing software** is **replaced** by **Stubs** and **Drivers** and **simulate the interface** between the software components in a simple manner



- Let's take an example to understand it in a better way.
- Suppose there is an application consisting of three modules say, **module A**, **module B** & **module C**.
- Developer has design in such a way that module **B depends on** module **A** & module **C depends** on module **B**
- The developer has **developed** the **module B** and now **wanted to test** it.
- **But** the module **A** and module **C** has **not** been **developed** yet.
- In that case **to test** the **module B completely** we can **replace** the module **A by Driver** and module **C by stub**

Diver & Stub (Unit Testing) Cont.

Driver

- **Driver** and/or **Stub** software **must be developed** for each **unit test**
- A **driver** is nothing more than a **"main program"**
  - It accepts test case data
  - Passes such data to the component and
  - Prints relevant results.
- **Driver**
  - Used in Bottom up approach
  - Lowest modules are tested first.
  - Simulates the higher level of components
  - Dummy program for Higher level component

Stub

- **Stubs** serve to replace **modules** that are subordinate (called by) the component to be tested.
- A **stub** or **"dummy subprogram"**
  - Uses the subordinate module's interface
  - May do minimal data manipulation
  - Prints verification of entry and
  - Returns control to the module undergoing testing
- **Stubs**
  - Used in Top down approach
  - Top most module is tested first
  - Simulates the lower level of components
  - Dummy program of lower level components

Integration Testing

Integration testing is the **process of testing** the **interface between two software units** or modules

It can be done in 3 ways	1. Big Bang Approach	2. Top Down Approach	3. Bottom Up Approach

Big Bang Approach

- **Combining all** the **modules** once and **verifying** the functionality after completion of individual module testing

Top Down Approach

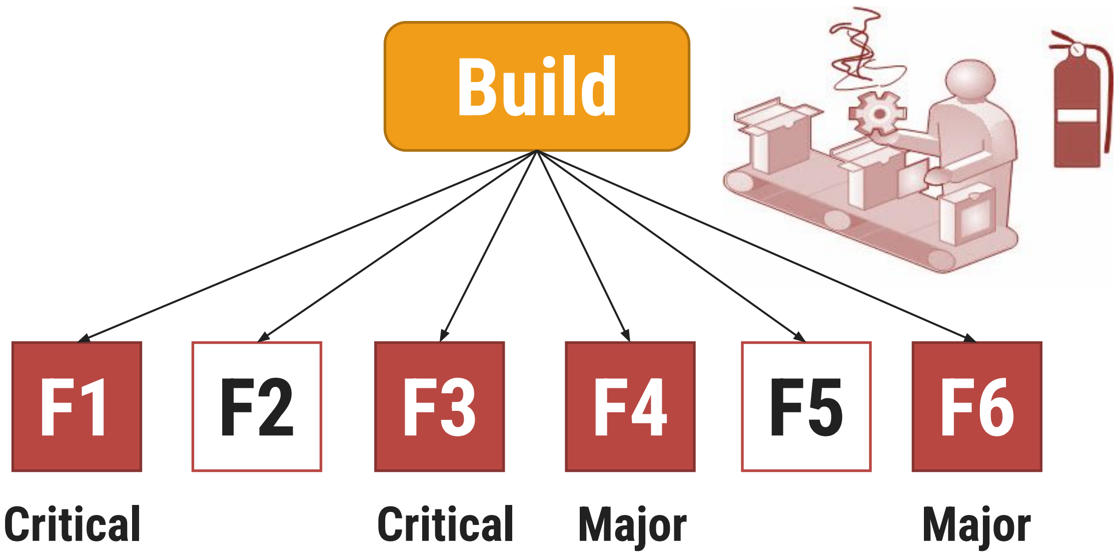
- **Testing take place** from **top** to **bottom**
- **High level** modules are **tested first** and then low-level modules and **finally integrated** the **low level modules** to high level to ensure the system is working as intended
- **Stubs** are used as a temporary module, if a module is not ready for integration testing

Bottom Up Approach

- **Testing take place** from **bottom** to **up**
- **Lowest level** modules are **tested first** and then high-level modules and finally integrated the high level modules to low level to ensure the system is working as intended
- **Drivers** are used as a temporary module, if a module is not ready for integration testing

Smoke Testing

- **Smoke Testing** is an integrated testing approach that is commonly used when product software is developed
- This test is performed after each Build Release
- Smoke testing verifies – Build Stability
- This testing is performed by “Tester” or “Developer”
- This testing is executed for Integration Testing, System Testing & Acceptance Testing
- What to Test?
  - All **major and critical functionalities** of the application is tested
  - It **does not go into depth** to test each functionalities
  - This does **not incudes detailed testing** for the build



It **test** the build **just to check** if any major or critical functionalities are broken

If there are smoke or Failure in the build after Test, build is rejected and developer team is reported with the issue

Validation Testing

- The **process** of evaluating software to **determine** whether it **satisfies specified business requirements** (client's need).
- It **provides** final **assurance** that **software meets** all informational, functional, behavioral, and performance **requirements**
- When **custom software** is build for one customer, a **series of acceptance tests** are conducted to validate all requirements
- It is **conducted** by **end user** rather then software engineers
- If **software** is developed as **a product** to be **used** by **many customers**, it is impractical to perform formal acceptance tests with each one
- Most software product builders use a process called **alpha** and **beta testing** to **uncover errors** that only the end user seems able to find

Validation Testing – Alpha & Beta Test

Alpha Test

- The alpha test is **conducted at the developer's site** by a **representative** group of **end users**
- The software is used in a **natural setting** with the **developer** "*looking over the shoulders*" of the **users** and **recording errors** and usage **problems**
- The alpha tests are **conducted** in a **controlled environment**

Beta Test

- The beta test is **conducted** at one or more **end-user sites**
- **Developers** are **not** generally **present**
- **Beta test** is a "*live*" **application of the software** in an environment that can **not** be **controlled by** the **developer**
- The **customer records** all **problems** and **reports** to the **developers** at regular intervals
- **After modifications**, software is **released** for **entire customer** base

System Testing

- In system testing the **software** and **other system elements** are **tested**.
- To test computer software, you spiral out in a clockwise direction along streamlines that increase the scope of testing with each turn.
- System testing **verifies that all elements mesh properly** and **overall system function**/performance is **achieved**.
- System testing is actually a **series of different tests** whose primary purpose is to fully exercise the computer-based system.

Types of System Testing

1. Recovery Testing
2. Security Testing
3. Stress Testing
4. Performance Testing
5. Deployment Testing

Types of System Testing

Recovery Testing

- It is a system test that **forces** the **software to fail** in a **variety of ways** and verifies that recovery is properly performed.
- **If recovery is automatic** (performed by the system itself)
  - **Re-initialization**, check pointing mechanisms, data recovery, and restart are evaluated for correctness.
- **If recovery requires human intervention**
  - **The mean-time-to-repair (MTTR) is evaluated** to determine whether it is within acceptable limits

Security Testing

- It **attempts** to **verify** software's **protection mechanisms**, which protect it from improper penetration (access).
- During this test, the **tester plays** the **role** of the individual who desires to **penetrate the system**.

Types of System Testing Cont.

Stress Testing

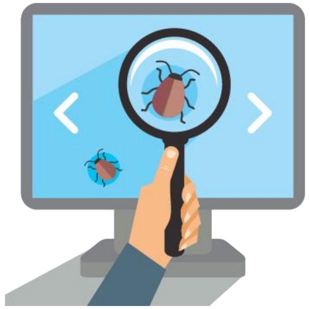
- It **executes a system** in a manner that **demands** resources in abnormal quantity, frequency or volume.
- A variation of stress testing is a technique called sensitivity testing.

Performance Testing

- It is designed to test the **run-time performance** of software.
- It occurs **throughout all steps** in the testing process.
- Even at the unit testing level, the performance of an individual module may be tested.

Types of System Testing Cont.

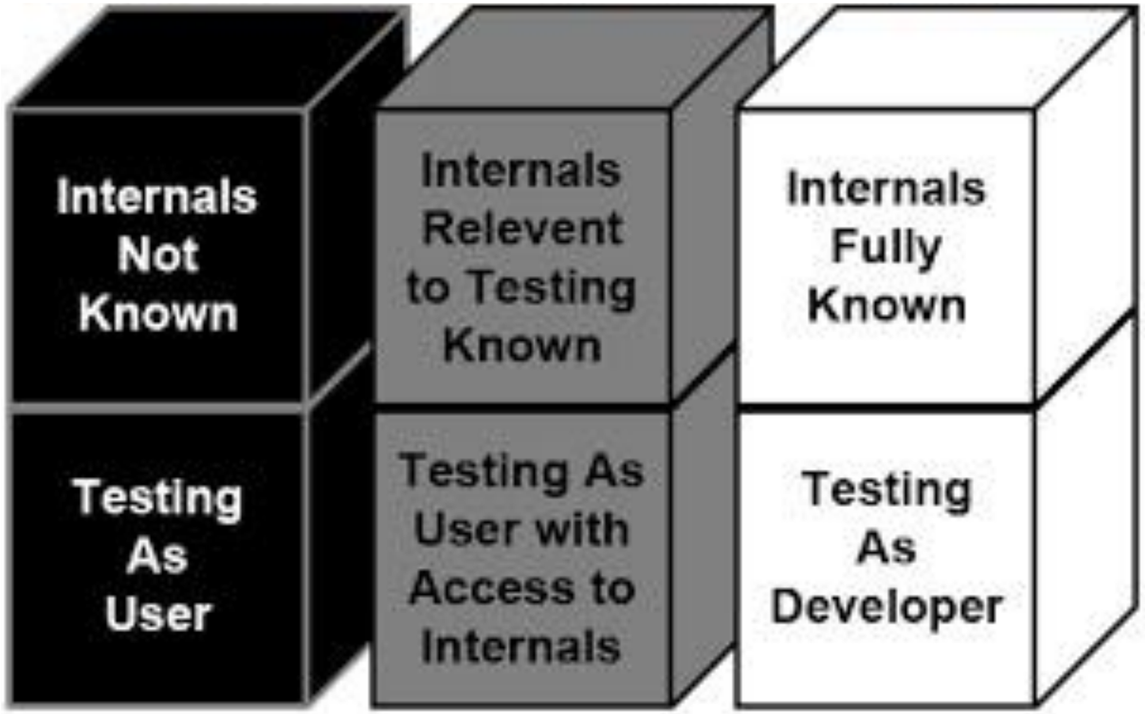
Deployment Testing



- It **exercises** the **software** in **each environment** in which it is **to operate**.
- In addition, it **examines**
  - All installation procedures
  - Specialized installation software that will be used by customers
  - All documentation that will be used to introduce the software to end users

Views of Test Objects

Black Box Testing	White Box Testing	Grey Box Testing
Close Box Testing Testing based only on specification	Open Box Testing Testing based on actual source code	Partial knowledge of source code


Internals Not Known
Testing As User
Internals Relevent to Testing Known
Testing As User with Access to Internals
Internals Fully Known
Testing As Developer

Black Box Testing

- Also known as **specification-based testing**
- **Tester** has **access** only to **running code** and the **specification** it is supposed to satisfy
- **Test cases** are **written with no knowledge of internal workings** of the code
- **No access** to **source code**
- So **test cases don’t worry** about **structure**
- **Emphasis** is only on **ensuring** that the **contract is met**

Advantages

- **Scalable;** not dependent on size of code
- Testing **needs no knowledge** of **implementation**
- **Tester** and **developer** can be **truly independent** of each other
- **Tests** are **done** with **requirements in mind**
- Does **not excuse inconsistencies** in the **specifications**
- **Test cases** can be **developed** in **parallel** with **code**

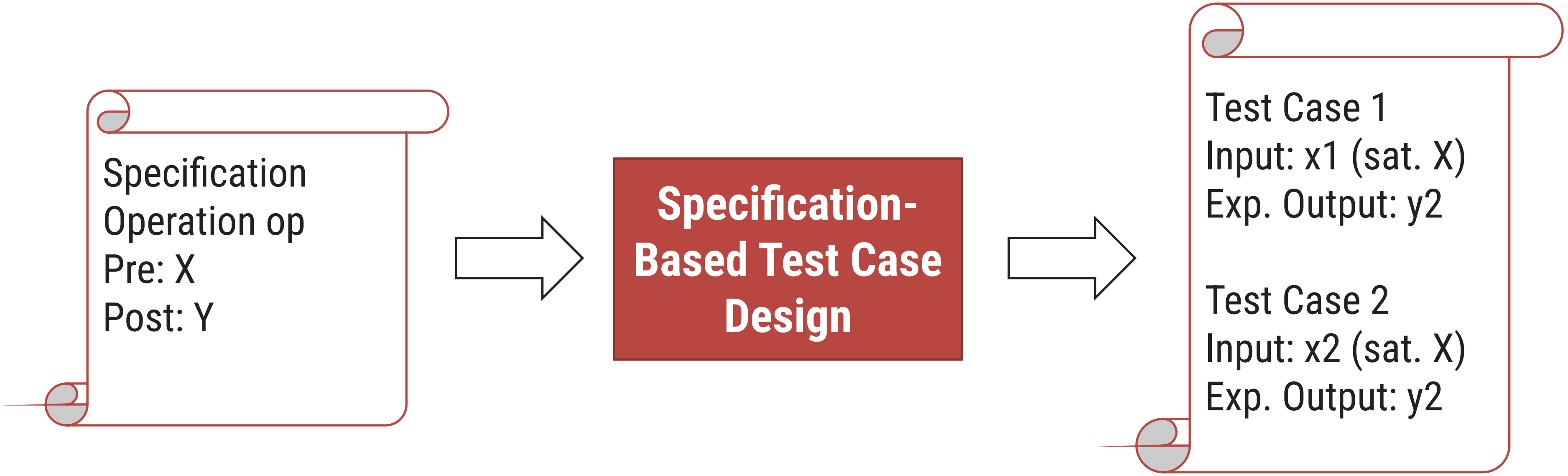
Black Box Testing Cont.

Disadvantages

- **Examine pre-condition**, and **identify** equivalence **classes**
- **All possible inputs** such that all classes **are covered**
- **Apply** the **specification to input** to write down **expected output**

Test Case Design

- **Test size** will **have to be small**
- **Specifications** must be **clear**, **concise**, and **correct**
- May **leave** many **program paths untested**
- **Weighting** of program **paths** is **not possible**



Specification
Operation op
Pre: X
Post: Y

Specification-Based Test Case Design

Test Case 1
Input: x1 (sat. X)
Exp. Output: y2

Test Case 2
Input: x2 (sat. X)
Exp. Output: y2

Black Box Testing Cont.

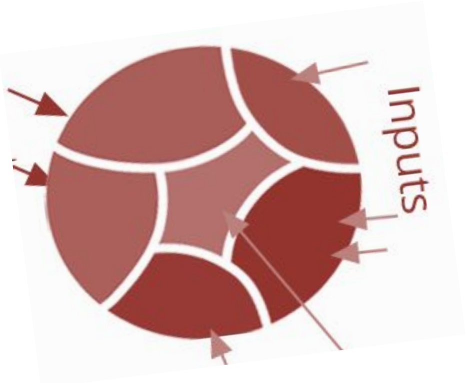
- **Exhausting testing** is not always possible when there is **a large set of input combinations**, because of **budget** and **time** constraint.
- The special techniques are needed which **select test-cases smartly** from the **all combination of test-cases** in such a way that all scenarios are covered

Two techniques are used

1 Equivalence Partitioning

2 Boundary Value Analysis (BVA)

Equivalence Partitioning



- **Input data** for a program unit usually **falls into** a **number of partitions**, e.g. all negative integers, zero, all positive numbers
- **Each partition** of input data **makes the program behave** in a **similar way**
- **Two test cases** based on members from the **same partition** is likely to **reveal the same bugs**

Equivalence Partitioning (Black Box Testing)

By **identifying** and **testing** *one member of each partition* we gain ***'good'*** coverage with ***'small'*** number of test cases

Testing one member of a **partition** should **be as good as testing any member** of the partition

Example - Equivalence Partitioning

For binary search the **following partitions** exist

- **Inputs** that *conform* to *pre-conditions*
- **Inputs** where the *precondition* is *false*
- **Inputs** where the *key element* is *a member* of the *array*
- **Inputs** where the *key element* is *not a member* of the *array*

- **Pick specific conditions** of the array
  - The array has a single value
  - Array length is even
  - Array length is odd

Equivalence Partitioning (Black Box Testing) Cont.

Example - Equivalence Partitioning

- Example: Assume that we have to test field which accepts SPI (Semester Performance Index) as input (SPI range is 0 to 10)

SPI ▭ * Accepts value 0 to 10

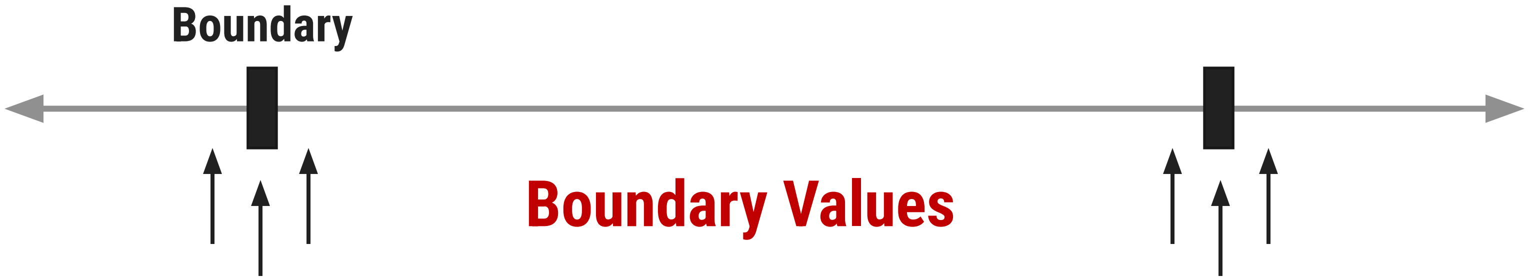
Equivalence Partitioning		
Invalid	Valid	Invalid
<=-1	0 to 10	>=11

- **Valid Class:** 0 – 10, pick any one input test data from 0 to 10
- **Invalid Class 1:** <=-1, pick any one input test data less than or equal to -1
- **Invalid Class 2:** >=11, pick any one input test data greater than or equal to 11

Boundary Value Analysis (BVA) (Black Box Testing)

- It arises from the **fact that most program fail at input boundaries**
- Boundary testing is the **process** of **testing between extreme ends** or boundaries between partitions of the input values.
- In Boundary Testing, Equivalence Class Partitioning plays a good role
- **Boundary Testing** comes after the **Equivalence Class Partitioning**
- The basic idea in boundary value testing is to **select input variable values at their**:

Just below the minimum	**Minimum**	Just above the minimum
Just below the maximum	**Maximum**	Just above the maximum



Boundary Value Analysis (BVA) (Black Box Testing)

- Suppose system asks for “a number between **100** and **999** **inclusive**”
- The **boundaries** are **100** and **999**
- We therefore **test for values**

99 100 101	**999 999 1000**
Lower boundary	Upper boundary

BVA - Advantages

- The BVA is **easy to use and remember** because of the uniformity of identified tests and the automated nature of this technique.
- One can **easily control the expenses** made on the testing by controlling the number of identified test cases.
- BVA is the **best approach** in cases where the **functionality** of a software is based on numerous variables representing physical quantities.
- The technique **is best at user input troubles** in the software.
- The **procedure and guidelines are crystal clear** and easy when it comes to determining the test cases through BVA.
- The **test cases** generated through BVA are **very small**.

Boundary Value Analysis (BVA) (Black Box Testing) Cont.

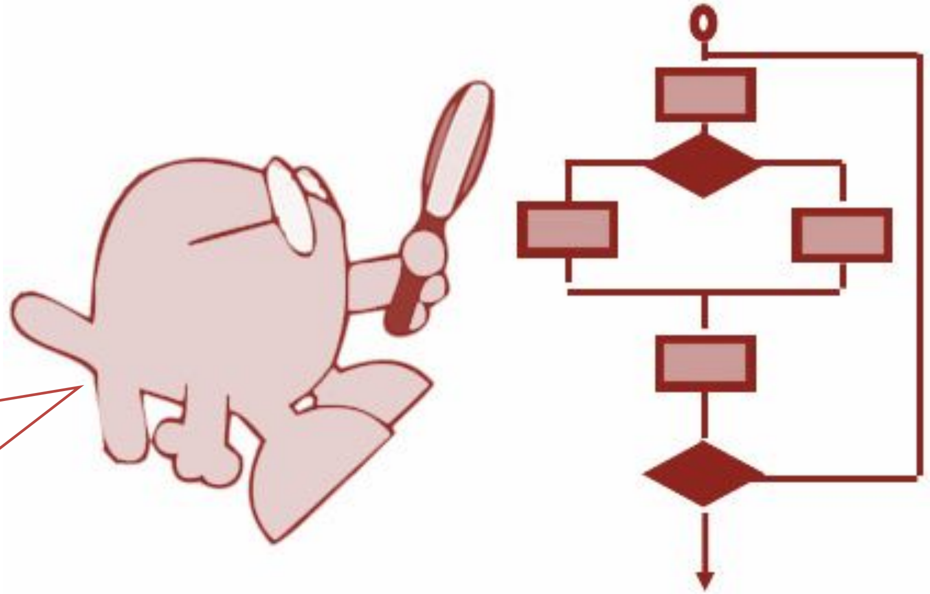
BVA - Disadvantages

- This technique **sometimes fails** to test **all the potential input** values. And so, the **results are unsure**.
- The **dependencies** with **BVA** **are not tested between two inputs**.
- This technique **doesn’t fit** well when it comes to **Boolean Variables**.
- It **only works** well with **independent variables** that depict quantity.

White Box Testing

- Also known as **structural testing**
- White Box Testing is a software testing **method** in which the **internal structure/design/implementation** of the module being tested **is known to the tester**
- Focus is on **ensuring** that even **abnormal invocations** are **handled gracefully**
- Using white-box testing methods, you can **derive test cases** that
  - **Guarantee** that all **independent paths** within a module have been **exercised at least once**
  - **Exercise** all **logical decisions** on their true and false sides
  - **Execute** all **loops** at their boundaries
  - **Exercise internal data structures** to ensure their validity

...our goal is to ensure that all statements and conditions have been executed at least once ...



White Box Testing Cont.

- **It is applicable to the following levels of software testing**
  - **Unit Testing:** For testing *paths within a unit*
  - **Integration Testing:** For testing *paths between units*
  - **System Testing:** For testing *paths between subsystems*

Advantages

- **Testing** can be **commenced** at an **earlier stage** as one need not wait for the GUI to be available.
- **Testing** is **more thorough**, with the possibility of **covering most paths**

Disadvantages

- Since **tests** can be very **complex**, **highly skilled resources** are **required**, with thorough **knowledge** of programming and implementation
- **Test script maintenance** can be a **burden**, if the implementation changes too frequently
- Since this method of testing is closely tied with the application being testing, tools to cater to every kind of implementation/platform may not be readily available

White-box testing strategies

- One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors.

White-box testing strategies

1 **Statement** coverage

2 **Branch** coverage

3 **Path** coverage

Statement coverage

- It aims to design test cases so that **every statement in a program is executed at least once**
- Principal idea is unless a statement is executed, it is very hard to determine if an error exists in that statement
- Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc.

White-box testing strategies Cont.

Consider the Euclid’s **GCD computation** algorithm

```
int compute_gcd(x, y)
int x, y;
{
    while (x! = y){
    if (x>y) then
          x= x – y;
    else y= y – x;
    }
    return x;
}
```

By choosing the test set **{(x=3, y=3), (x=4, y=3), (x=3, y=4)}**, we can exercise the program such that all statements are executed at least once.

White-box testing strategies Cont.

Branch coverage

- In the branch coverage based testing strategy, **test cases are designed to make each branch condition to assume true and false values in turn**
- It is also known as **edge Testing** as in this testing scheme, each edge of a program's control flow graph is traversed at least once
- Branch coverage **guarantees statement coverage**, so it is stronger strategy compared to Statement Coverage.

Path Coverage

- In this strategy test cases are executed in such a way that **every path is executed at least once**
- All possible control paths taken, including
  - **All loop paths** taken **zero**, **once** and **multiple items** in technique
  - The **test case** are **prepared** based on the **logical complexity measure** of the procedure design
- **Flow graph**, **Cyclomatic Complexity** and **Graph Metrices** are used to arrive at basis path.

Grey Box Testing

- **Combination** of **white box** and **black box** testing
- **Tester** has **access** to **source code**, but uses it **in** a **restricted manner**
- **Test cases** are still **written using specifications** based on expected outputs for given input
- These **test cases are informed** by **program code structure**